[SKT AI Course: Deep Learning Basics]

TensorFlow Basics

From "CS224n: Natural Language Processing with Deep Learning"
(Created by Chip Huyen)

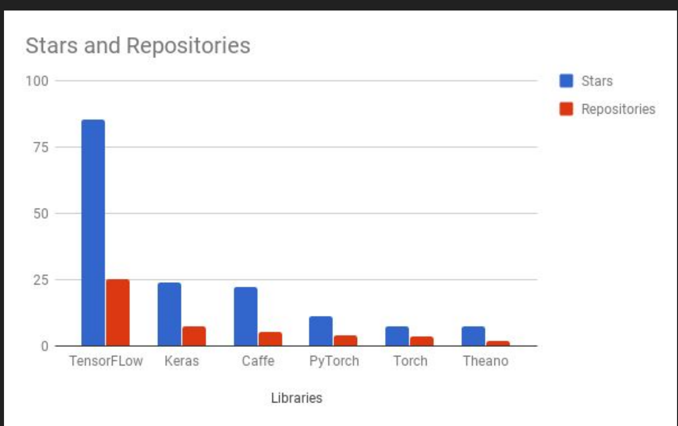
TA: Eunji Jun
Instructor: Heung-Il Suk
http://milab.korea.ac.kr

Department of Brain and Cognitive Engineering,
Korea University

May 27, 2019

Why TensorFlow?

- Flexibility + Scalability
- Popularity

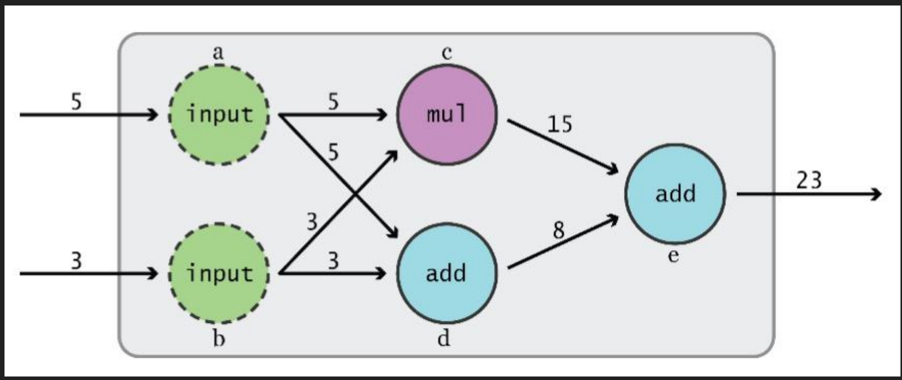


```
import tensorflow as tf
```


Graphs and Sessions

Data Flow Graphs

TensorFlow separates definition of computations from their execution

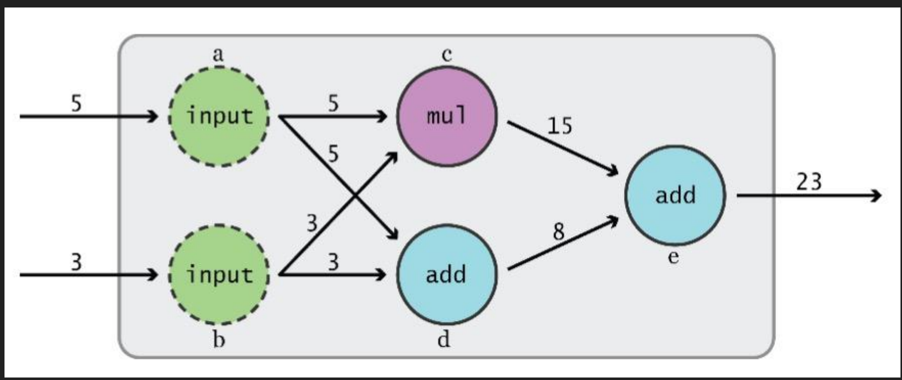


Graph from *TensorFlow for Machine Intelligence*

Data Flow Graphs

Phase 1: assemble a graph

Phase 2: use a session to execute operations in the graph.



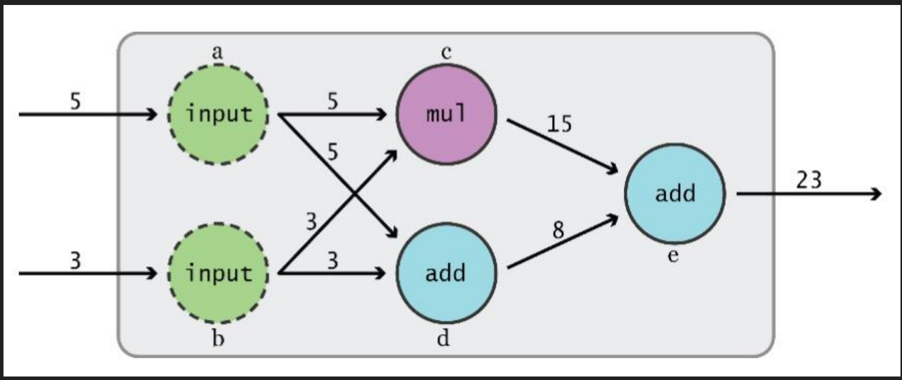
Graph from *TensorFlow for Machine Intelligence*

Data Flow Graphs

Phase 1: assemble a graph

This might change in the future with eager mode!!

Phase 2: use a session to execute operations in the graph.



Graph from *TensorFlow for Machine Intelligence*

What’s a tensor?

What's a tensor?

An n-dimensional array

0-d tensor: scalar (number)

1-d tensor: vector

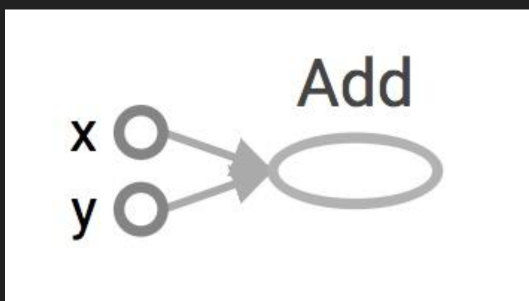
2-d tensor: matrix

and so on

Data Flow Graphs

```
import tensorflow as tf
a = tf.add(3, 5)
```

Visualized by TensorBoard



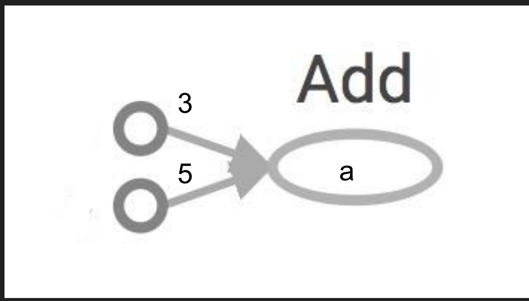
Why x, y?

TF automatically names the nodes when you don't explicitly name them.
x = 3
y = 5

Data Flow Graphs

```
import tensorflow as tf
a = tf.add(3, 5)
```

Interpreted?

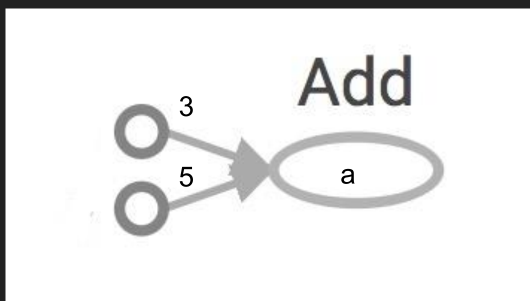


Nodes: operators, variables, and constants
Edges: tensors

Tensors are data.
TensorFlow = tensor + flow = data + flow
(I know, mind=blown)

Data Flow Graphs

```
import tensorflow as tf
a = tf.add(3, 5)
print(a)
```




```
>> Tensor("Add:0", shape=(), dtype=int32)
```
(Not 8)

How to get the value of a?

Create a **session**, assign it to variable sess so we can call it later

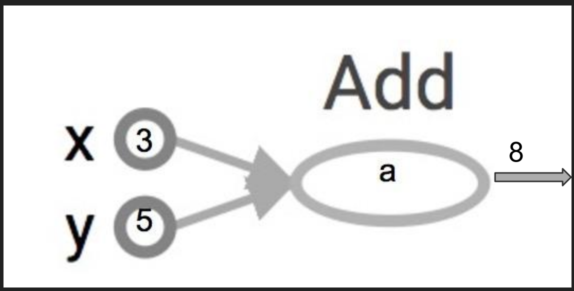
Within the session, evaluate the graph to fetch the value of a

How to get the value of a?

Create a **session**, assign it to variable sess so we can call it later

Within the session, evaluate the graph to fetch the value of a

```
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
print(sess.run(a))        >> 8
sess.close()
```



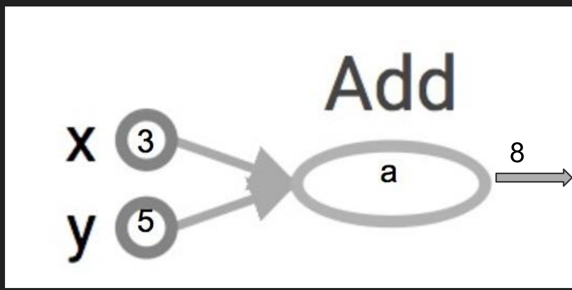
The session will look at the graph, trying to think: hmm, how can I get the value of a, then it computes all the nodes that leads to a.

How to get the value of a?

Create a **session**, assign it to variable sess so we can call it later

Within the session, evaluate the graph to fetch the value of a

```
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
with tf.Session() as sess:
     print(sess.run(a))
sess.close()
```



tf.Session()

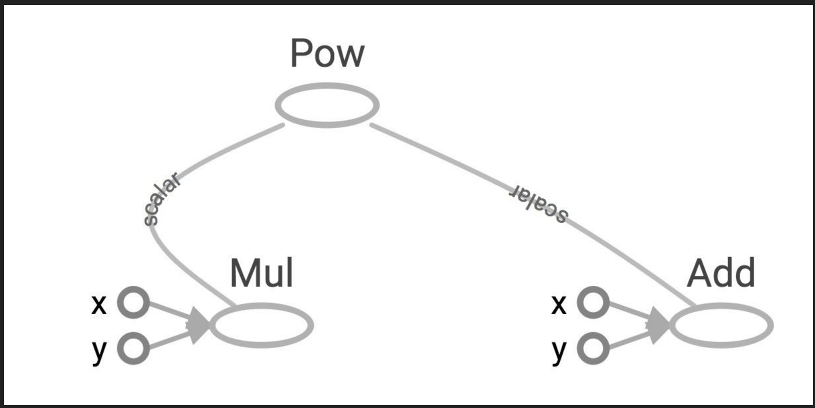
A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.

Session will also allocate memory to store the current values of variables.

More graph

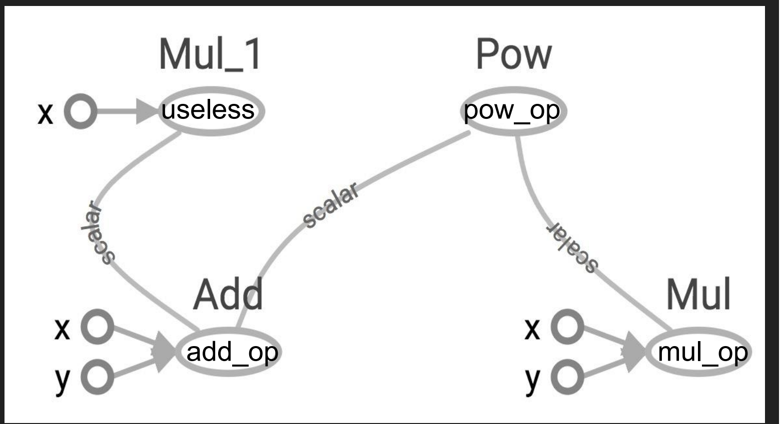
Visualized by TensorBoard

```
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
      op3 = sess.run(op3)
```



Subgraphs

```
x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
      z = sess.run(pow_op)
```

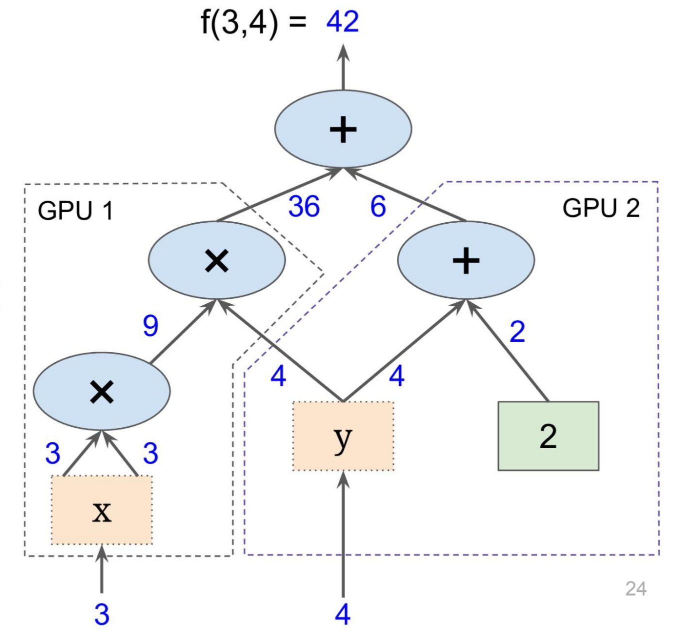


Because we only want the value of pow_op and pow_op doesn't depend on useless, session won't compute value of useless
→ save computation

Subgraphs

Possible to break graphs into several chunks and run them parallelly across multiple CPUs, GPUs, TPUs, or other devices

Example: AlexNet



Graph from *Hands-On Machine Learning with Scikit-Learn and TensorFlow*

Distributed Computation

To put part of a graph on a specific CPU or GPU:

```
# Creates a graph.
with tf.device('/gpu:2'):
  a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='a')
  b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='b')
  c = tf.multiply(a, b)

# Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

# Runs the op.
print(sess.run(c))
```

Why graphs

1. Save computation. Only run subgraphs that lead to the values you want to fetch.
2. Break computation into small, differential pieces to facilitate auto-differentiation
3. Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices
4. Many common machine learning models are taught and visualized as directed graphs

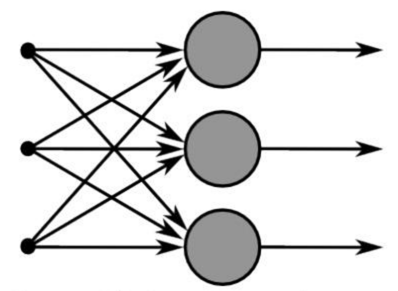


Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input x.

A neural net graph from Stanford's CS224N course

TensorBoard

Visualize it with TensorBoard

```
import tensorflow as tf

a = tf.constant(2, name='a')
b = tf.constant(3, name='b')
x = tf.add(a, b, name='add')

writer = tf.summary.FileWriter('./graphs', tf.get_default_graph())
with tf.Session() as sess:
     # writer = tf.summary.FileWriter('./graphs', sess.graph)
     print(sess.run(x))
writer.close() # close the writer when you’re done using it
```

Create the summary writer after graph definition and before running your session

‘graphs’ or any location where you want to keep your event files

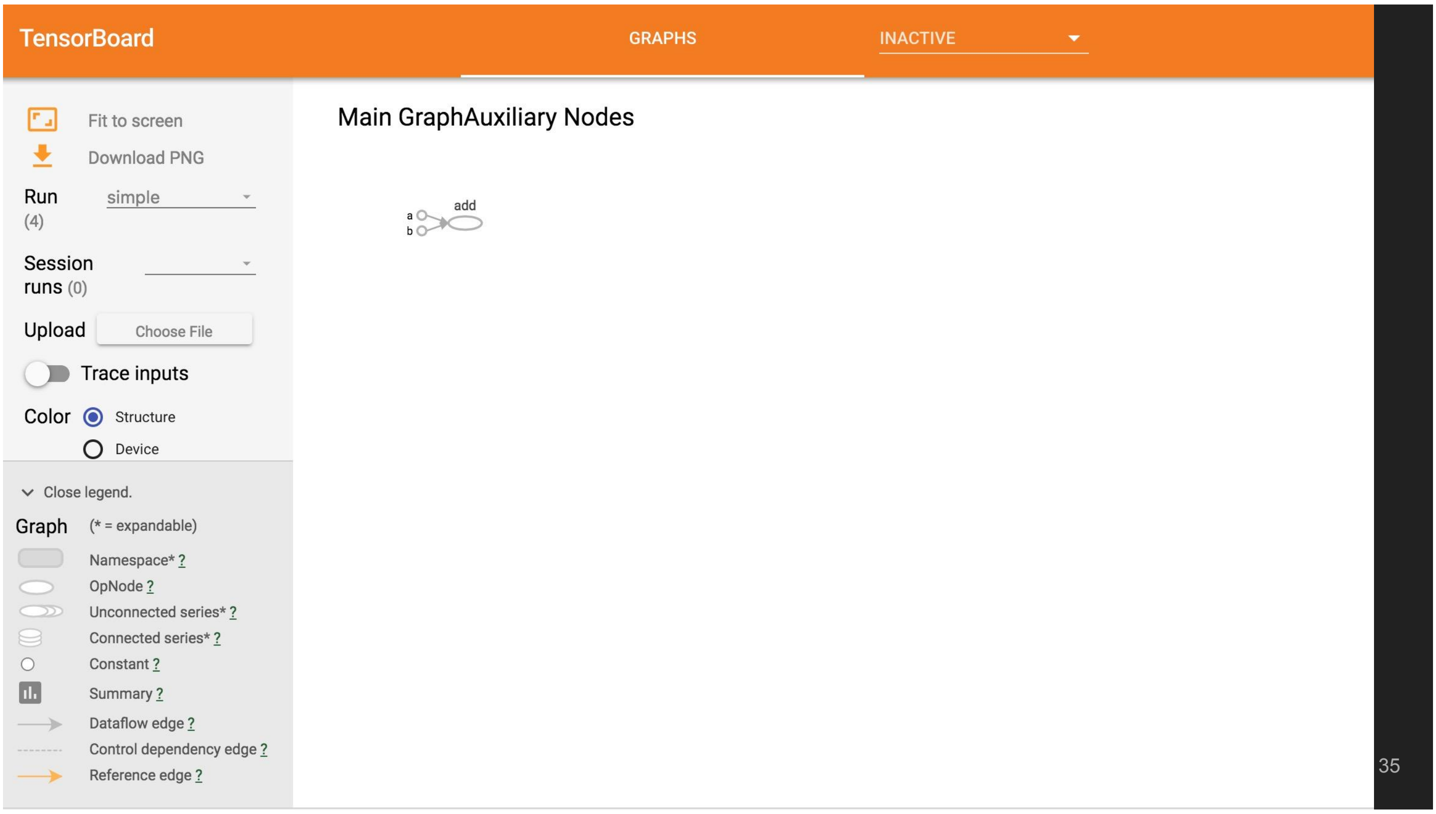
Run it

Go to terminal, run:

```
$ python [yourprogram].py
$ tensorboard --logdir="./graphs" --port 6006
```

6006 or any port you want

Then open your browser and go to: `http://localhost:6006/`



Constants, Sequences, Variables, Ops

Constants

```
import tensorflow as tf

a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')
x = tf.multiply(a, b, name='mul')

with tf.Session() as sess:
     print(sess.run(x))

#  >>  [[0 2]
#       [4 6]]
```

Broadcasting similar to NumPy

Tensors filled with a specific value

```
tf.zeros([2, 3], tf.int32) ==> [[0, 0, 0], [0, 0, 0]]

# input_tensor is [[0, 1], [2, 3], [4, 5]]

tf.zeros_like(input_tensor) ==> [[0, 0], [0, 0], [0, 0]]

tf.fill([2, 3], 8) ==> [[8, 8, 8], [8, 8, 8]]
```

Similar to NumPy

Constants as sequences

tf.lin_space(start, stop, num, name=None)

```
tf.lin_space(10.0, 13.0, 4) ==> [10. 11. 12. 13.]
```

tf.range(start, limit=None, delta=1, dtype=None, name='range')

```
tf.range(3, 18, 3) ==> [3 6 9 12 15]
tf.range(5) ==> [0 1 2 3 4]
```

NOT THE SAME AS NUMPY SEQUENCES

Tensor objects are not iterable

```
for _ in tf.range(4): # TypeError
```


Randomly Generated Constants

```
tf.random_normal
tf.truncated_normal
tf.random_uniform
tf.random_shuffle
tf.random_crop
tf.multinomial
tf.random_gamma
```

Randomly Generated Constants

```
tf.set_random_seed(seed)
```


TF vs NP Data Types

TensorFlow integrates seamlessly with NumPy

```
tf.int32 == np.int32            # ⇒ True
```

Can pass numpy types to TensorFlow ops

```
tf.ones([2, 2], np.float32)     # ⇒ [[1.0 1.0], [1.0 1.0]]
```

For **`tf.Session.run(fetches)`**: if the requested fetch is a Tensor , output will be a NumPy ndarray.

```
sess = tf.Session()
a = tf.zeros([2, 3], np.int32)
print(type(a))                  # ⇒ <class 'tensorflow.python.framework.ops.Tensor'>
a_out = sess.run(a)
print(type(a))                  # ⇒ <class 'numpy.ndarray'>
```

Use TF DType when possible

- Python native types: TensorFlow has to infer Python type
- NumPy arrays: NumPy is not GPU compatible

What’s wrong with constants?

Not trainable

Constants are stored in graph definition

```
my_const = tf.constant([1.0, 2.0], name="my_const")

with tf.Session() as sess:
     print(sess.graph.as_graph_def())
```

```
attr {
  key: "value"
  value {
    tensor {
      dtype: DT_FLOAT
      tensor_shape {
        dim {
          size: 2
        }
      }
      tensor_content: "\000\000\200?\000\000\000@"
    }
  }
}
```


Constants are stored in graph definition

This makes loading graphs expensive when constants are big

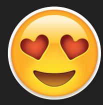
Only use constants for primitive types.

Use variables or readers for more data that requires more memory

Variables

```
# create variables with tf.Variable
s = tf.Variable(2, name="scalar")
m = tf.Variable([[0, 1], [2, 3]], name="matrix")
W = tf.Variable(tf.zeros([784,10]))
```

```
# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(2))
m = tf.get_variable("matrix", initializer=tf.constant([[0, 1], [2, 3]]))
W = tf.get_variable("big_matrix", shape=(784, 10), initializer=tf.zeros_initializer())
```



You have to <u>initialize</u> your variables

The easiest way is initializing all variables at once:

```
with tf.Session() as sess:
	sess.run(tf.global_variables_initializer())
```

Initializer is an op. You need to execute it within the context of a session

You have to <u>initialize</u> your variables

The easiest way is initializing all variables at once:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

Initialize only a subset of variables:

```
with tf.Session() as sess:
    sess.run(tf.variables_initializer([a, b]))
```


You have to <u>initialize</u> your variables

The easiest way is initializing all variables at once:

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
```

Initialize only a subset of variables:

```
with tf.Session() as sess:
    sess.run(tf.variables_initializer([a, b]))
```

Initialize a single variable

```
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
    sess.run(W.initializer)
```

Eval() a variable

```
# W is a random 700 x 100 variable object
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
      sess.run(W.initializer)
      print(W)

>> Tensor("Variable/read:0", shape=(700, 10), dtype=float32)
```


tf.Variable.assign()

```
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
     sess.run(W.initializer)
     print(W.eval())                  # >> 10
```

W.assign(100) creates an assign op.
That op needs to be executed in a session
to take effect.

tf.Variable.assign()

```
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
     sess.run(W.initializer)
     print(W.eval())                    # >> 10

--------

W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
     sess.run(W.initializer)
     sess.run(assign_op)
     print(W.eval())                    # >> 100
```


Placeholder

A quick reminder

A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.

Placeholders

A TF program often has 2 phases:
1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Assemble the graph first without knowing the values needed for computation

Analogy:
Define the function $f(x, y) = 2 * x + y$ without knowing value of x or y.
x, y are placeholders for the actual values.

Why placeholders?

We, or our clients, can later supply their own data when they need to execute the computation.

Placeholders

tf.placeholder(dtype, shape=None, name=None)

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

with tf.Session() as sess:
      print(sess.run(c))                 # >> InvalidArgumentError: a doesn't an actual value
```

Supplement the values to placeholders using a dictionary

Placeholders

tf.placeholder(dtype, shape=None, name=None)

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

with tf.Session() as sess:
      print(sess.run(c, feed_dict={a: [1, 2, 3]}))    # the tensor a is the key, not the string ‘a’

# >> [6, 7, 8]
```

Placeholders

`tf.placeholder(dtype, shape=None, name=None)`

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])

b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # short for tf.add(a, b)

with tf.Session() as sess:
      print(sess.run(c, feed_dict={a: [1, 2, 3]}))

# >> [6, 7, 8]
```

Quirk:
shape=None means that tensor of any shape will be accepted as value for placeholder.

shape=None is easy to construct graphs and great when you have different batch sizes, but nightmarish for debugging

Placeholders

`tf.placeholder(dtype, shape=None, name=None)`

```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # Short for tf.add(a, b)

with tf.Session() as sess:
      print(sess.run(c, {a: [1, 2, 3]}))

# >> [6, 7, 8]
```

Quirk:
shape=None also breaks all following shape inference, which makes many ops not work because they expect certain rank.

Placeholders are valid ops

```
tf.placeholder(dtype, shape=None, name=None)
```

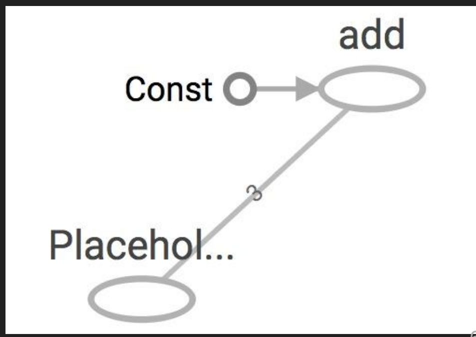
```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b  # Short for tf.add(a, b)

with tf.Session() as sess:
      print(sess.run(c, {a: [1, 2, 3]}))

# >> [6, 7, 8]
```



What if want to feed multiple data points in?

You have to do it one at a time

```
with tf.Session() as sess:
      for a_value in list_of_values_for_a:
            print(sess.run(c, {a: a_value}))
```

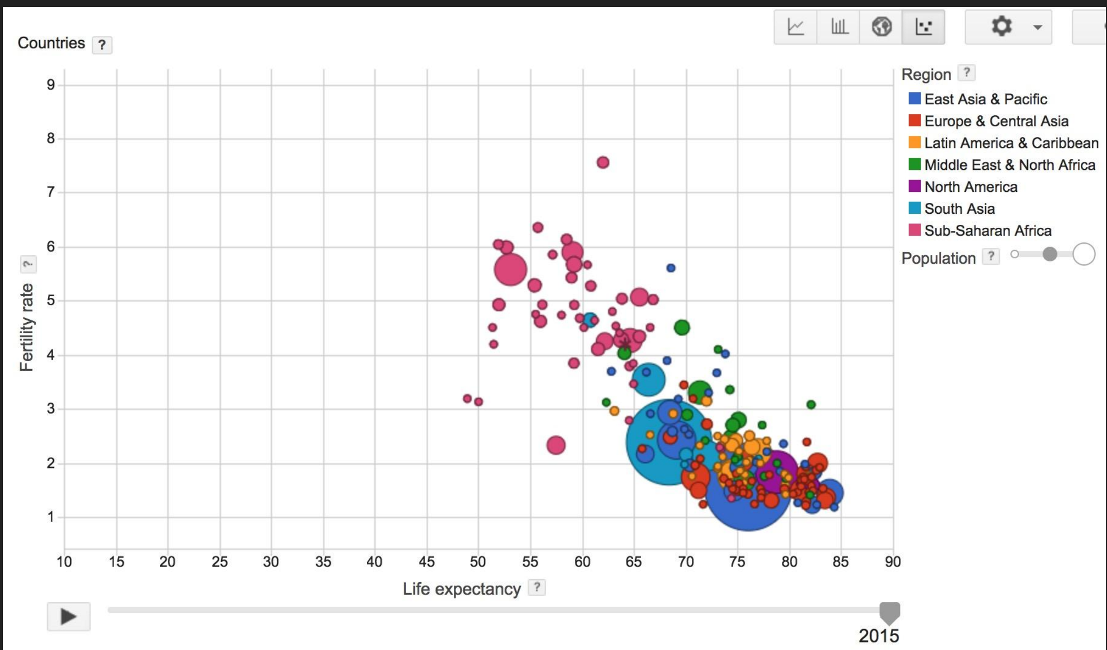
Extremely helpful for testing

Feed in dummy values to test parts of a large graph

Linear Regression in TensorFlow

Model the linear relationship between:

- dependent variable Y
- explanatory variables X



Visualization made by Google, based on data collected by World Bank

World Development Indicators dataset

X: birth rate
Y: life expectancy
190 countries

Want

Find a linear relationship between X and Y
to predict Y from X

Model

Inference: Y_predicted = w * X + b

Mean squared error: $E[(y - y\_predicted)^2]$

Interactive Coding

birth_life_2010.txt

Interactive Coding

`linreg_starter.py`

`birth_life_2010.txt`

Phase 1: Assemble our graph

Step 1: Read in data

I already did that for you

Step 2: Create placeholders for inputs and labels

```
tf.placeholder(dtype, shape=None, name=None)
```

Step 3: Create weight and bias

```
tf.get_variable(
    name,
    shape=None,
    dtype=None,
    initializer=None,
    …
)
```

No need to specify shape if using constant initializer

Step 4: Inference

```
Y_predicted = w * X + b
```

Step 5: Specify loss function

```
loss = tf.square(Y - Y_predicted, name='loss')
```


Step 6: Create optimizer

```
opt = tf.train.GradientDescentOptimizer(learning_rate=0.001)

optimizer = opt.minimize(loss)
```

Phase 2: Train our model

Step 1: Initialize variables

Step 2: Run optimizer

(use a feed_dict to feed data into X and Y placeholders)

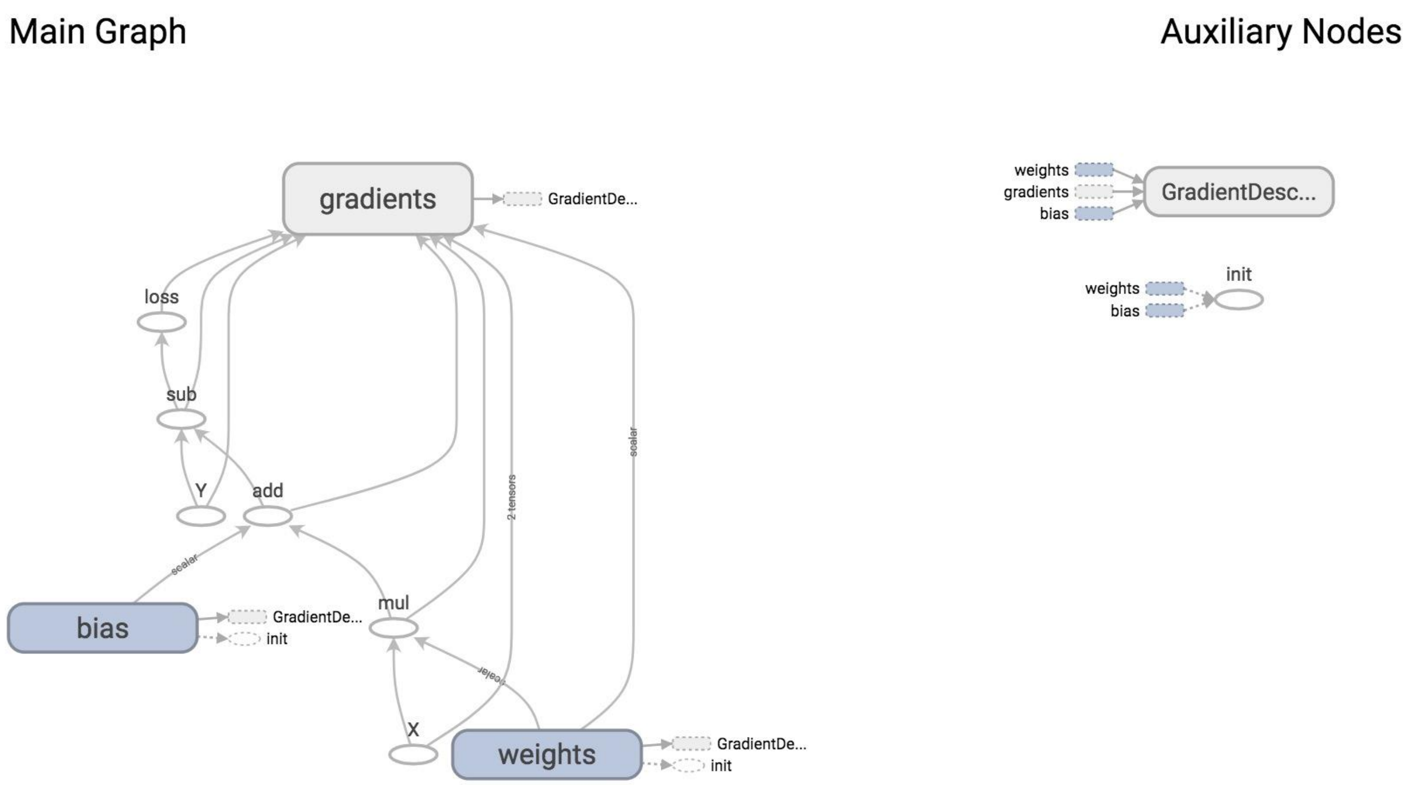
Write log files using a FileWriter

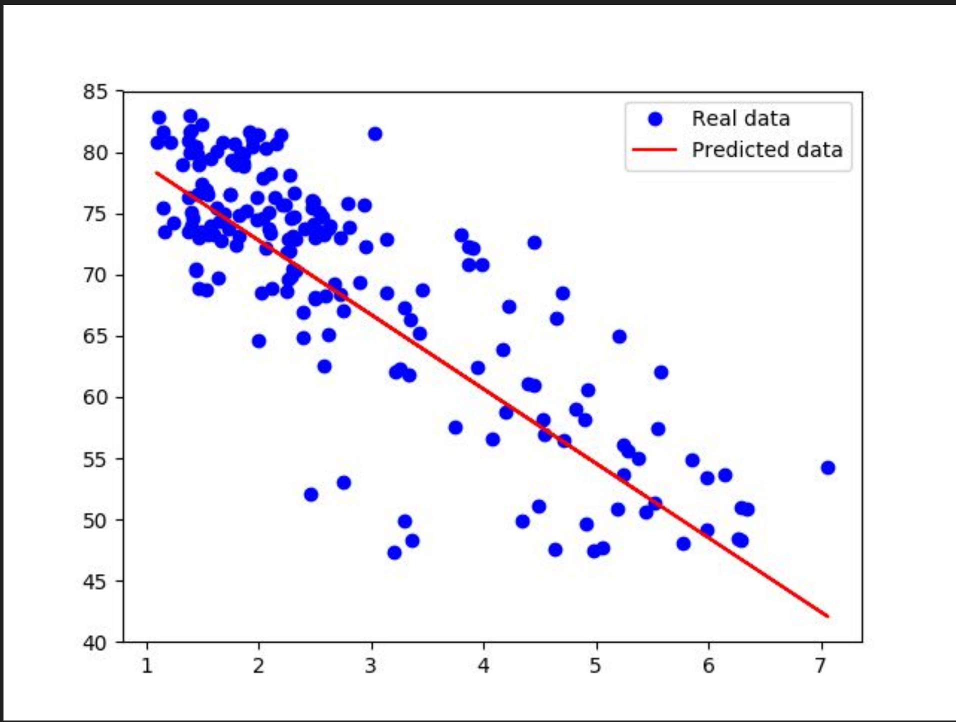
```
writer = tf.summary.FileWriter('./graphs/linear_reg', sess.graph)
```

See it on TensorBoard

```
Step 1: $ python linreg_starter.py

Step 2: $ tensorboard --logdir='./graphs'
```



tf.data

Placeholder

Pro: put the data processing outside TensorFlow, making it easy to do in Python

Cons: users often end up processing their data in a single thread and creating data bottleneck that slows execution down.

Placeholder

```
data, n_samples = utils.read_birth_life_data(DATA_FILE)

X = tf.placeholder(tf.float32, name='X')
Y = tf.placeholder(tf.float32, name='Y')
…
with tf.Session() as sess:
       …
     # Step 8: train the model
     for i in range(100): # run 100 epochs
          for x, y in data:
               # Session runs train_op to minimize loss
               sess.run(optimizer, feed_dict={X: x, Y:y})
```

tf.data

Instead of doing inference with placeholders and feeding in data later, do inference directly with data

tf.data

tf.data.Dataset

tf.data.Iterator

Store data in tf.data.Dataset

- `tf.data.Dataset.from_tensor_slices((features, labels))`
- `tf.data.Dataset.from_generator(gen, output_types, output_shapes)`

Store data in tf.data.Dataset

```
tf.data.Dataset.from_tensor_slices((features, labels))

dataset = tf.data.Dataset.from_tensor_slices((data[:,0], data[:,1]))
```

Store data in tf.data.Dataset

```
tf.data.Dataset.from_tensor_slices((features, labels))

dataset = tf.data.Dataset.from_tensor_slices((data[:,0], data[:,1]))

print(dataset.output_types)      # >> (tf.float32, tf.float32)

print(dataset.output_shapes)     # >> (TensorShape([]), TensorShape([]))
```


Can also create Dataset from files

- `tf.data.TextLineDataset(filenames)`
- `tf.data.FixedLengthRecordDataset(filenames)`
- `tf.data.TFRecordDataset(filenames)`

tf.data.Iterator

Create an iterator to iterate through samples in Dataset

tf.data.Iterator

- `iterator = dataset.make_one_shot_iterator()`
- `iterator = dataset.make_initializable_iterator()`

tf.data.Iterator

- `iterator = dataset.make_one_shot_iterator()`
  Iterates through the dataset exactly once. No need to initialization.
- `iterator = dataset.make_initializable_iterator()`
  Iterates through the dataset as many times as we want. Need to initialize with each epoch.

tf.data.Iterator

```
iterator = dataset.make_one_shot_iterator()
X, Y = iterator.get_next()         # X is the birth rate, Y is the life expectancy

with tf.Session() as sess:
     print(sess.run([X, Y]))          # >> [1.822, 74.82825]
     print(sess.run([X, Y]))          # >> [3.869, 70.81949]
     print(sess.run([X, Y]))          # >> [3.911, 72.15066]
```

tf.data.Iterator

```
iterator = dataset.make_initializable_iterator()

...

for i in range(100):
        sess.run(iterator.initializer)
        total_loss = 0
        try:
            while True:
                sess.run([optimizer])
        except tf.errors.OutOfRangeError:
            pass
```


Handling data in TensorFlow

```
dataset = dataset.shuffle(1000)

dataset = dataset.repeat(100)

dataset = dataset.batch(128)

dataset = dataset.map(lambda x: tf.one_hot(x, 10))
# convert each element of dataset to one_hot vector
```

Does tf.data really perform better?

Does tf.data really perform better?

With placeholder: 9.05271519 seconds

With tf.data: 6.12285947 seconds

Should we always use tf.data?

- For prototyping, feed dict can be faster and easier to write (pythonic)
- tf.data is tricky to use when you have complicated preprocessing or multiple data sources
- NLP data is normally just a sequence of integers. In this case, transferring the data over to GPU is pretty quick, so the speedup of tf.data isn't that large

How does TensorFlow know what variables to update?

Optimizers

Optimizer

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(loss)

_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```

Optimizer

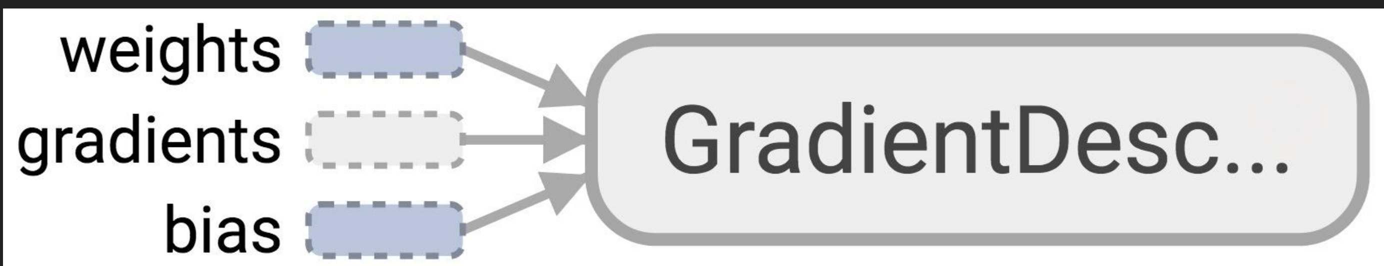
```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)

_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```

Session looks at all trainable variables that loss depends on and update them

Optimizer

Session looks at all trainable variables that optimizer depends on and update them



Trainable variables

```
tf.Variable(initial_value=None, trainable=True,...)
```

Specify if a variable should be trained or not
By default, all variables are trainable

List of optimizers in TF

```
tf.train.GradientDescentOptimizer

tf.train.AdagradOptimizer

tf.train.MomentumOptimizer

tf.train.AdamOptimizer

tf.train.FtrlOptimizer

tf.train.RMSPropOptimizer

...
```

Usually Adam works out-of-the-box better than SGD